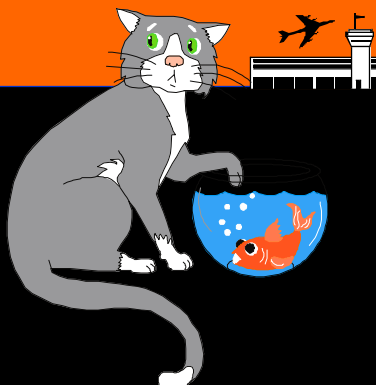


**CSI**  
*Serving the aircraft and CNS/ATM communities*



**C++**  
*The natural enemy of DO-178B?*

**Mike DeWalt**  
**Chief Scientist, Aviation Systems**  
**Certification Services, Inc.**  
**[mike.dewalt@certification.com](mailto:mike.dewalt@certification.com)**  
**Voice +1.360.376.8110**

**2002 FAA Software Conference**

C++ - 1

Copyright © 2002 Certification Services, Inc.

## 2002 FAA Software Conference



- C++ should never be used in a safety critical system (hatten95)
- Results of safety forum on C++
  - 26% favored use
  - 26% undecided
  - 30% would not consider
  - 17% use is dangerous



- C++ is well known and has vocal advocates and detractors
- Significant correspondence to other languages
- Provides good case study for goal
- *Goal: establish criteria to judge any language used in civil certification*

C++ has enjoyed enormous growth and support. Every computer science graduate is familiar with C++. The language is seemingly everywhere. It has also been maligned by some of the biggest names in the safety community.

The reason I picked C++ for this presentation is that C++ is easily recognized and most people have heard at least some controversy surrounding it. I want to use C++ as an example of how computer programming languages could be handled within a DO-178B environment. The principles we should use to judge C++ can be used for any language. Although other language advocates might protest, C++ has significant features in common with other popular (and supposedly safer) languages such as Ada and Pascal. The goal is to establish under what conditions **any** given language is suitable for use on civil certification projects.

CSI
Safety forum conclusions

Against	For
<ul style="list-style-type: none"> <li>■ <i>Predictable execution can't be reasoned from source code</i></li> <li>■ <i>Memory leak not statically determinable</i></li> <li>■ <i>Order of execution not defined</i></li> <li>■ <i>Behavior of STL uncertain</i></li> <li>■ <i>Safe subset doesn't exist</i></li> </ul>	<ul style="list-style-type: none"> <li>■ <i>Availability of tools and experience</i></li> <li>■ <i>Safe subset is possible</i></li> <li>■ <i>No worse than other large languages (Ada)</i></li> <li>■ <i>Safety incidents not correlated to language/CMM level</i></li> <li>■ <i>Requirements better target</i></li> </ul>

Copyright © 2002 Certification Services, Inc.
2002 FAA Software Conference
C++ - 4

The European Railway signaling standard raised the possibility of using C++ in its safety-critical applications. Brian Wichmann, a respected authority on programming languages from the National Physical Laboratory in the UK, warned of dangers in such use.

Wichmann started a discussion of C++ on the Internet (<http://www.cs.york.ac.uk/hise/sclist/cplussafety.html>) involving about 20 key people in software and safety (Leveson, Mellor, McDermid, et al). As might be expected, there was considerable opposition to C++. Much of the criticism centered on potential problems rather than specific evidence of any direct safety impact. As Les Hatton points out in *Safer C* (McGraw-Hill, 1995), a large proportion of errors in programs tend to remain undiscovered for a long time. He also makes the point that most of these are discoverable by static analysis via tools. However, his stated position is that C++ shouldn't be used for safety-critical systems.

On the "for" side, a strong case was made that languages have little to do with eventual safety when compared to system design, requirements, software design robustness, and other properties not related to language choice. Most safety-critical systems adopt restricted subsets of whatever language is used. Ada, as used on the Boeing 777 fly-by-wire flight controls, was touted as an example of this during the discussion thread.

Trying to evaluate a language in a vacuum, while with some merit, doesn't provide the needed information from a safety viewpoint. The real issue is whether the behavior of the system can be reasoned from the source code in conjunction with all of the other artifacts from DO-178B. If so, then DO-178B objectives provide an acceptable level of confidence that the specified behavior (including derived requirements) and the execution behavior are equivalent.



- **begin{**
  - Determine language-sensitive objectives and guidance
  - Determine desired features
  - Evaluate applicable features for ability to meet objectives
  - Establish error-prone portions of language
  - Produce enforceable coding standards to govern language subset and usage
- **}end**

There are many claims as to why one should or should not use C++. There is little to help you determine systematically if any given language -- assembly, Pascal, Ada, whatever -- is suitable for safety-critical applications. However, DO-178B provides a lens through which to view a proposed language. If all of DO-178B's objectives at a given criticality can be satisfied using the proposed programming language, then that language can be used for code development at that criticality.

Such an evaluation hinges on the objectives and additional guidance that could be affected by language choice. The resulting criteria should be suitable for judging any language.

Most projects seek a subset of language features based on project need. The subset is then evaluated against the criteria referred to above. Language features that fail to meet the criteria are rejected. Rejection might be due to difficulty of verification, to unpredictable behavior, or to behavior that is known to be unacceptable.

Once the basic language has been reduced to an approvable subset, there might be other issues associated with features that can be misused or misunderstood. Such features have a high probability of introducing errors into the final product. This is more subjective, and different organizations will come up with different answers.



- Sec. 4.4.2: Language and compiler considerations
  - Product verification validates compiler for that product
  - Planning considers language and compiler
  - Verification considers language and compiler
- Language-sensitive objectives
  - A1: plans, environment, considerations (+tools), standards,
  - A5: all
  - A7: Structural Coverage/D&CC
  - A10: understanding, means of compliance
- Issue: reasoning about behavior from source code

Section 4.4.2, Language and Compiler Considerations, has the following wording: Upon successful completion of verification of the software product, the compiler is considered acceptable for that product. For this to be valid, the software verification process activities need to consider particular features of the programming language and compiler. The software planning process considers these features when choosing a programming language and planning for verification.

The following objectives are affected to a lesser or greater degree by language choice:

Table A-1

- 1 Software development and integral processes activities are defined. 4.1a, 4.3 – additional activities may be required (e.g., structural coverage feasibility, training, reviews of checklists). Checklists used for other languages might be inappropriate for C++ and could require changes.
- 3 Software life cycle environment is defined. 4.1c
- 4 Additional considerations are addressed. 4.1d – If the language is uncommon in avionics, this section should contain proposed rationale and mitigations.
- 5 Software development standards are defined. 4.1e – these will follow from the language analyses discussed earlier.

Table A-5: all; While there should be little problem in reviewing the source code against the design, the choice of programming languages might not fit well with the design description language. This could require extra direction to reviewers or changes to the design description approach.

Table A-7: 5 through 6; structural coverage (modified condition/decision, 6.4.4.2; decision coverage), 6.4.4.2a 6.4.4.2b; statement coverage 6.4.4.2a 6.4.4.2b data coupling and control coupling 6.4.4.2c. The concern is whether the proposed coverage approach provides sufficient confidence in the executable image. This is a concern for constructs such as templates, class definitions, inheritance, and polymorphism. This would be less of a problem at Level A. For Level A, the activity performed to identify added functionality added by the compiler (e.g object to source code correspondence analysis) should identify any unexpected behaviors.

Table A-10

- 1 Communication and understanding between the applicant and the certification authority is established. 9.0
- 2 The means of compliance is proposed

If the language is already familiar to the certification authority or if there has been substantial avionics experience with the language, then little needs to be said other than identify the language. However, in the case of C++, there is little familiarity or experience.



- DO-248B
  - FAQ 31: Verification of product
  - DP 12: Source to object code
  - DP 13: Structural coverage
- Cast 8 issue paper: Use of C++ language
- FAA OOTia

DO-248B has no specific guidance on acceptability of computer languages. It does mention the effect of language choice. FAQ 31 explains why the testing of the final product validates the compiler for that specific instantiation, relieving the applicant of compiler qualification. DP 12 discusses the effects of language choice on functionality added by the compiler. DP 13 discusses the effects of language features on structural coverage analysis.

CAST has released their issue paper on use of C++. The paper identifies issues to be addressed by users of C++ but provides no guidance on acceptability. There was spirited discussion of these issues at the 2002 FAA/NASA conference on Object Oriented Technology in Aviation. The conference was well attended and probed deeply into object-oriented technology. Although the conference was intended to be language-independent, many of the issues and approaches were directly applicable to C++.



- Summary
  - Determine features
  - Capability to meet objectives
  - Evaluate “error-proneness”
  - Develop coding standards
- Use existing (safe?) subsets
  - Defined by language experts
  - Project usefulness
  - Validation of results
- Roll your own





- NIST
- QA C++ (based on Safer C principles)
  - Commercial tool support
- EC ++
- Other homegrown definitions

Unlike Ada, which has often had accepted subsets, C++ has not converged on any widely accepted safe subset. NIST has developed a set of guidelines for use of C++, as has the Nuclear Regulatory Agency (NRC). In addition to C++ the NRC has proposed guidelines for using other programming languages as well.

A company in England (see references at the end of this presentation) has developed a tool tied to the company's definition of a safe subset of C++. This company had produced an earlier tool tied to Hatton's *Safer C* rules. The newer tool follows similar lines.

On a more public scale, an embedded specification for C++ has been developed and recently approved by a committee constituted for this purpose. The specification's subset attempts to ensure predictable execution. Several restrictions are aimed at run-time efficiency.

Other, smaller efforts can be found on the web.



## ■ Union

- ISO: 9.5 (1 page)
- Issue: Unstructured type conversion undefined
- Solution: Prohibit unjustified use/create safe union

## ■ Exceptions

- ISO: Section 15 (6 pages)
- An issue: Invalid object state
- A solution: Invariant assured

Union – This feature (also available in Pascal as a variant record). This allows multiple types to be overlayed on the same memory location. See section 9.5 of the standard. This is a method of saving memory by reusing it for different data providing the programmer can keep track of which variable has the latest stored value. It can be used to provide low level conversion of types however this will be machine and compiler implementation dependent. For example there may be two huge arrays used in a program that are sequentially stored and read in a manner that they will never interfere with each other (i.e. write a[], read a[], write b[], read b[]). To save memory they can be made into a union. However changes to timing dependent parameters could result in inconsistent states if the reading and writing overlapped between the two arrays. While there are safe constructs for doing this simply requiring each use to be justified and evaluated for safety during reviews is probably the best approach if limited usage is made of this construct.

Exceptions – These are a built – in C++ approach to handling run time errors (also available in Ada). See section 15 of the standard. Exceptions are invasively wound into the run time environment and the compiler logic. The difficulty of dealing with exceptions belies the 6 pages in the ISO standard (50+ pages were devoted to templates a much less intrusive construct). One of the issues with using exceptions is that persistent object or global variable could be left in an invalid state. If objects contain invariants, these can be checked prior to throwing an exception and ensuring that the invariant is maintained. For example code for a class will not create partially populated or un-initialized objects. This can create additional complexity for code which can be traded off against potential predictable error handling. Other issues are created with the use of exceptions and placement syntax since standard resource allocation and de-allocation will not be used typically (ISO 15.2. Interestingly Embedded C++ does not include exceptions and the NIST document discourages the use of exceptions. Compiler implementation of this feature has not been consistent according to the NIST document.



- **Multiple inheritance**
  - ISO: 10.1 (1 page)
  - An issue: duplicate names in different ancestors create ambiguity
  - Solution: Create overloaded function specifying base class

Multiple Inheritance– The idea is that characteristics from a number of base classes can be combined into a subclass. This is discussed thoroughly in the OOTia position paper on inheritance. This is one issue from the general theory that is actually dealt with by the ANSI Standard. While the standard requires the compiler to reject this as ill formed, this could result in redesign on a large project with deep and widely distributed class hierarchies. One approach is to provide standards entries that that require disambiguation. A more elegant approach will require the use of explicitly overloaded base class definitions which specify the name of the desired base class. Here is a case where the compiler behavior is well specified but standards prevent potential rework. EC++ eliminates the multiple inheritance feature and the NIST implementation strongly discourages its use. If it is used NIST puts strong limitations on its application.



- Reference types: implicit modification
- ++ operator, especially array assignment
  - `A[i] = A[i] + i++;`
  - `Y=Y+++C;`
- Confusing `==` with `=`
- Documented implicit casting  
`mdbl=mchar+mint`

The issue of error-proneness is difficult. In the case of the ++ operator, the first example shown is undefined by the standard since order of operation is undefined. The outcome is implementation-dependent. Intimate knowledge of the compiler and testing of all occurrences of this construct could alleviate the problem. The second example shown is specified by the language but little known and poorly understood. It is easily handled by requiring white space around every token:

`Y^=^Y^+^ ++^C^` (where ^ represents white space)

Some constructs might be desirable in spite of their potentially troublesome natures. A given piece of code containing a deprecated construct might be more confusing without the construct. In such cases, the developer should try to mitigate the construct's undesirable aspects through appropriate coding standards, reviews, additional verification, and so on. An example is the "equal" confusion shown in the slide above. Some organizations have attempted -- without success -- to solve this by using

```
#define IsEqualTo ==
```

whereas the use of C++ rules, placing constants on the left of comparisons, or targeted reviews might be more effective.

While the C++ standard mostly defines the effect of the following formula

```
doubleType = charType + integerType + doubleType,
```

repeated application of implicit typing in different program sections may result in an unexpected result. The easiest approach is to disallow all implicit type conversion except by conversion. Another approach is to define rules for when it is allowed and why.

Experience matters. A shop with six years of deep C++ experience will make different mistakes than a shop adopting the language for the first time.



- Establish language definition
- Determine desired constructs



- Accepted standard or reference
  - ISO/IEC 14882 (1 September 1998)
  - Embedded C++
- Specific implementation
  - Microsoft C++
  - Borland C++
  - GNU C++
  - Avionics embedded compilers

The C++ standard was released on 1 September 1998. Not all C++ compilers comply with the standard. Both the standard and the implementation must be evaluated. There is often no way to guarantee consistent execution behavior from one compiler to the next.

The ISO standard for C++ is about 715 pages, the Ada language reference manual about 650.

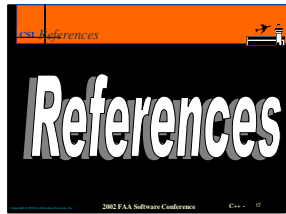


- Used to establish conformance to a “formally” defined language
- Normally not an issue for DO-178B
- Could be used in selection of language
- Issue of service experience vs validation



- DO-178B provides a means to establish acceptable use of a specific language
- Language analysis is difficult and time-consuming
- Any language, including C++, can be adapted to DO-178B environment





### Regulatory information

- 1) “Use of the C++ Programming Language”, Certification Authorities Software Team (CAST) Position Paper CAST-8, January, 2002
- 2) OOTia – <http://shemesh.larc.nasa.gov/foot/> discussed thoroughly in other presentations

### Safe subsets and related information

- 1) “The Use of the C++ Programming Language for the Development of Safety-Critical Systems”, Nigel G. Backhurst, November 1995
- 2) “C++ in Safety-Critical Systems”, David W. Binkley, NIST, November 1995
- 3) QA C++ tool-defined safe subset,  
[http://www.programmingresearch.com/solutions/generic\\_cpp.htm](http://www.programmingresearch.com/solutions/generic_cpp.htm)
- 4) “Embedded C++: an overview”, P.J. Plauger, Embedded Systems Programming, December 1997
- 5) “Rationale for the Embedded C++ Specification”, Version WP-RA -003, Embedded C++ Technical Committee, 1998
- 6) “Review Guidelines for Software Written in High Level Programming Language Used in Safety Systems”  
NUREG CR/6463 Rev. 1, 1997 [Editorial – this reference doesn’t provide a real subset but more of things to examine in different languages including C++]

### Standards

- 1) “Embedded C++ Specification”, Version WP-AM-003, Embedded C++ Technical Committee, October 1999
- 2) “Programming Languages C++”, ISO/IEC 14882, September 1998

### Other Information

- 1) “Safer C: Developing Software for High-integrity and Safety-Critical Systems”, Les Hatton, McGraw-Hill Book Company, 1995
- 2) Moderated discussion on C++ and Safety, Brian Wichmann,  
<http://www.cs.york.ac.uk/hise/sclist/cplussafety.html>

## 2002 FAA Software Conference